PyTrilinos Users Guide
Development Branch

Bill Spotz
wfspotz@sandia.gov
Sandia National Laboratories

28 November 2009

Version: Trilinos 10.0, PyTrilinos 4.2

Abstract

PyTrilinos is a python interface to selected Trilinos packages. The Trilinos Project is a collection of over 30 software packages written primarily in C++ that provide linear-, nonlinear-, and eigen-solvers, along with preconditioners and supporting utilities, that are object-oriented, parallel and serial, for sparse and dense problems. PyTrilinos is one of those packages, and provides python interfaces to the most popular and important Trilinos packages.

Contents

1 Introduction

PyTrilinos is a python interface to selected Trilinos packages. The Trilinos Project is a collection of over 30 software packages written primarily in C++ that provide linear-, nonlinear-, and eigen-solvers, along with preconditioners and supporting utilities, that are object-oriented, parallel and serial, for sparse and dense problems. PyTrilinos is one of those packages, and provides python wrappers to the following packages:

Package	Description
Teuchos	Fundamental tools package
Epetra	Linear algebra services
TriUtils	Testing utilities
EpetraExt	Extensions to Epetra
Pliris	Dense solver package
AztecOO	Iterative linear solvers
Galeri	Example processor maps and matrices
Amesos	Direct linear solvers

Package	Description
IFPACK	Incomplete factorization preconditioning
Komplex	Complex linear solvers
Anasazi	Eigensolvers
ML	Multi-level preconditioners
NOX	Nonlinear solvers
LOCA	Continuation algorithms (disabled)

There are a number of ways to obtain PyTrilinos documentation. You should start with the PyTrilinos Tutorial section and consult the Frequently Asked Questions web page for any initial questions.

Running python interactively, you can use the `help()` or `dir()` functions on PyTrilinos modules, classes, methods or objects. Within a UNIX shell, you can run `pydoc` on any class in the PyTrilinos hierarchy.

If the python documentation strings do not provide sufficient information, then you should consult this documentation, specifically the sections referring to individual PyTrilinos modules. If the class or method you are interested in does not appear within these sections, then use the Trilinos C++ documention, as the C++ and python interfaces are kept analogous whenever possible.

2 PyTrilinos Prerequisites

To build PyTrilinos, you must have the following installed:

- Python **2.3 or higher.** Some packages require that the python interpreter be able to distinguish between boolean and integer values. This support was first provided in python 2.3. PyTrilinos has been upgraded to work with python 2.6.
- **The** numpy **python module.** We recommend version 1.0.1 or higher, but backward compatibility has been maintained with versions 0.9.x.
- SWIG **1.3.39 or higher.** SWIG is the Simple Wrapper and Interface Generator, and is the workhorse for generating the python interface to Trilinos packages. As of this writing, version 1.3.39 is the current release version.

3 Building PyTrilinos

For release 10.0, all of Trilinos has converted from an autotools build system to a CMake build system. CMake can generate several different types of build systems, from the familiar Makefile systems, to Windows Visual Studio Pro systems and Mac OS X XCode systems. This obviously allows Trilinos an entry to Windows platforms. In addition, the autotools version of Trilinos did not support libtool, so the adoption of CMake provides a new and robust support for shared Trilinos libraries.

From the point of view of PyTrilinos, the move to CMake provides for robust and portable shared library support. Shared libraries have been required for PyTrilinos since release 9.0, so this is an important advance. Unfortunately, this portability does not yet extend to Windows, as this capability will require some additional work on all of Trilinos. Therefore, PyTrilinos is not yet available under Windows.

To build PyTrilinos, `cmake` should be run at the top build directory level with the options:

```
-D Trilinos_ENABLE_PyTrilinos:BOOL=ON
-D BUILD_SHARED_LIBS:BOOL=ON
```

Turning the shared libraries on explicitly is required because the Trilinos policy is to not turn on shared libraries implicitly. Turning PyTrilinos on will build python modules for every Trilinos package that has python wrappers defined. If you want to ensure that every available PyTrilinos module is built, you should use the option:

```
-D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES:BOOL=ON
```

This is currently the configuration used by PyTrilinos developers to test the package, so it should be considered a safe way to build PyTrilinos.

Previously, you had to set the environment variable `LD_LIBRARY_PATH` (`DYLD_LIBRARY_PATH` on Mac OS X) in order to run PyTrilinos tests. This requirement has been lifted with the adoption of CMake.

4 Known Issues

- **64-bit.** We routinely build PyTrilinos successfully on a 64-bit platform. PyTrilinos for Trilinos release 10.0 may also require that the third-party libraries must be linked as shared libraries as well, depending on the platform.
- **MPICH.** PyTrilinos has been successfully built and run using Open MPI (formerly LAM/MPI), but MPICH seems to give it problems. Apparently the MPICH version of `mpirun` adds command-line arguments to the invocation of the executable, and our python test scripts do not currently handle these properly.

5 PyTrilinos Tutorial

`PyTrilinos` is a collection of python modules that allows a python programmer to access selected `Trilinos` packages, either dynamically or within a script. For example, once `PyTrilinos` is installed, a user running python interactively who wanted to gain access to `Epetra` classes would type

```
>>> from PyTrilinos import Epetra
```

To import a `PyTrilinos` module, use the `from PyTrilinos import <module>` syntax, where `<module>` generally corresponds to the `Trilinos` namespace you want to access. (Note the `Epetra` C++ package does not use namespaces, and a design decision was made to strip the `Epetra_` prefix from `Epetra` classes and put them in a python namespace `Epetra`).

This tutorial will touch on the following packages:

- Epetra
- Teuchos
- EpetraExt
- TriUtils
- Amesos
- AztecOO
- ML

No package is completely wrapped. See the python `dir()` or `help()` function or the PyTrilinos web documentation for a list of those classes within each package that are wrapped.

`PyTrilinos` supports MPI, allowing for parallel python scripts. The `Epetra.PyComm()` function will return an `Epetra_MpiComm` communicator if `Trilinos` was built with MPI support, and an `Epetra_SerialComm` communicator otherwise. In addition, if `Trilinos` was configured with MPI support, then the `Epetra` module will internally call `MPI_Init()` when imported and register `MPI_Finalize()` with the `atexit` python module. Thus, scripts can be written for either environment transparently.

5.1 Epetra

To see what is available in the `Epetra` namespace imported above, do a

```
>>> dir(Epetra)
```

This will print a list of all the attributes (classes, functions and objects) in the namespace. Currently, you will get a list of well over 600 strings. Some of these will be familiar `Epetra` names (with the `Epetra_` prefix removed), such as `'SerialComm'`. A `SerialComm` communicator can be created with

```
>>> comm = Epetra.SerialComm()
```

This object has (almost) all the methods of the C++ `Epetra_SerialComm` class and is recognized internally by python as an `Epetra_SerialComm` object. The `dir()` function works on objects, too:

```
>>> dir(comm)
```

gives a (much shorter) list of the attributes for `comm`, including all of the methods that can be called on the object. For example,

```
>>> comm.NumProc()
1
>>> comm.MyPID()
0
>>> comm.Label()
'Epetra::Comm'
```

For more detailed help, use

```
>>> help(comm)
```

or access the documentation pages of the appropriate `Trilinos` package. `PyTrilinos` provides a wrapper for the `Epetra_MpiComm` class and introduces the `Epetra.PyComm()` function, that returns an `Epetra_MpiComm` object if `Trilinos` was configured with MPI support and an `Epetra_SerialComm` otherwise.

We can use `comm` to create an `Epetra_Map` (which provides support for distributing global indexes across processors) for a map with 9 elements, strting with index 0, using communicator `comm`, by typing

```
>>> map = Epetra.Map(9,0,comm)
```

Typically, `PyTrilinos` classes support a python `__str__()` method, which internally calls the object's `Print()` method (or `print` method, as appropriate) and is in turn used by the python `print` command:

```
>>> print map

Number of Global Elements  = 9
Number of Global Points = 9
Maximum of all GIDs        = 8
Minimum of all GIDs        = 0
Index Base                 = 0
Constant Element Size      = 1

Number of Local Elements   = 9
Number of Local Points  = 9
Maximum of my GIDs         = 8
Minimum of my GIDs         = 0
```

```
     MyPID          Local  Index        Global  Index
         0               0                   0
         0               1                   1
         0               2                   2
         0               3                   3
         0               4                   4
         0               5                   5
         0               6                   6
         0               7                   7
         0               8                   8
```

The `Print()` (or `print`) methods typically work by providing a python file object, default `sys.stdout`, where the C++ classes expect a stream.

We can now use `map` to create an `Epetra_Vector`:

```
>>> vect = Epetra.Vector(map)
```

The `Epetra.Vector` class actually inherits from both the `Epetra_Vector` C++ class and the `UserArray` python class from the `numpy` module. The constructors ensure that both base classes point to the same data buffer. Thus an `Epetra.Vector` has all the methods and capabilities of an `Epetra_Vector`, plus the capabilities of a python array:

```
>>> vect[4] = 3.14
>>> print vect
[ 0.    0.    0.    0.    3.14  0.    0.    0.    0.  ]
>>> vect.shape = (3,3)
>>> print vect
[[ 0.    0.    0.  ]
 [ 0.    3.14  0.  ]
 [ 0.    0.    0.  ]]
>>> vect[2,2] = 2.718
>>> print vect
[[ 0.     0.     0.   ]
 [ 0.     3.14   0.   ]
 [ 0.     0.     2.718]]
```

Epetra also allows the creation of (compressed row) sparse matrices. Let us consider here the definition of a matrix corresponding to the discretization of a 1D Laplace problem on a regular Cartesian grid. We store the matrix as an `Epetra.CrsMatrix`, whose constructor requires a map, and an estimation of the number of nonzeros per row (in this instance, 3):

```
>>> A = Epetra.CrsMatrix(Epetra.Copy,map,3)
```

The argument `Epetra.Copy` specifies CopyMode; see the Epetra documentation for more details.

We can now create the matrix, by adding one row at a time. For each row, `indices` contains global column indices, and `values` the corresponding values.

```
>>> numLocalRows = map.NumMyElements()
>>> for lid in range(numLocalRows):
...   gid = map.GID(lid)
...   if gid == 0:
...     indices = [gid, gid + 1]
...     values  = [2.0, -1.0   ]
...   elif gid == n - 1:
...     indices = [gid, gid - 1]
...     values  = [2.0, -1.0   ]
```

```
...   else:
...     indices = [gid, gid - 1, gid + 1]
...     values  = [2.0, -1.0   , -1.0   ]
...   A.InsertGlobalValues(lid, values, indices)
...
```

Finally, we transform the matrix representation into one based on local indices. The transformation is required in order to perform efficient parallel matrix-vector products and other matrix operations.

```
>>> ierr = A.FillComplete()
```

Once the matrix, the solution vector (`x`) and the right-hand side vector (`b`) have been created, it is convenient to store them in an `Epetra.LinearProblem` object, which can be created either as

```
>>> problem = Epetra.LinearProblem(A, x, b)
```

or as follows:

```
>>> problem = Epetra.LinearProblem()
>>> problem.SetOperator(A)
>>> problem.SetLHS(x)
>>> problem.SetRHS(b)
```

Methods `GetMatrix()`, `GetLHS()` and `GetRHS()` can be used to extract the linear system matrix, the solution vector, and the right-hand side vector, respectively.

5.2 Teuchos

The primary purpose for `PyTrilinos.Teuchos` is support for the `ParameterList` class, which is used by several `Trilinos` packages for setting solution parameters, flags and output behavior. Often, this is handled implicitly, as `PyTrilinos` has been designed to accept python dictionaries in place of `ParameterList` objects. However, you can build a `ParameterList` directly:

```
>>> from PyTrilinos import Teuchos
>>> pList = Teuchos.ParameterList()
>>> pList.set("maxiters", 100)
>>> pList.set("tol", 1.0e-6)
>>> pList.set("precond", "ILUT")
```

The python version of `ParameterList` is augmented to behave somewhat like dictionaries:

```
>>> for name in pList:
...   print name, ":", pList[name]
...
maxiters : 100
precond : ILUT
tol : 1.0e-6
```

You can convert a `ParameterList` to an `XMLObject`:

```
>>> writer = Teuchos.XMLParameterListWriter()
>>> xmlObj = writer.toXML(pList)
>>> print xmlObj
<ParameterList>
<Parameter name="maxiters" type="int" value="100"/>
<Parameter name="precond" type="string" value="ILUT"/>
<Parameter name="tol" type="double" value="1e-06"/>
</ParameterList>

>>> open("params.xml","w").write(xmlObj.toStr())
```

You can also read an XML ParameterList from disk:

```
>>> source = Teuchos.FileInputSource("params.xml")
>>> xmlObj = source.getObject()
>>> reader = Teuchos.XMLParameterListReader()
>>> pList  = reader.toParameterList(xmlObj)
```

5.3 EpetraExt

Module EpetraExt contains several utilities to read and write Epetra objects, in particular maps, vectors, and matrices.

An Epetra.Map object can be saved to a file using the command

```
>>> from PyTrilinos import EpetraExt
>>> filename = "map.mm"
>>> EpetraExt.BlockMapToMatrixMarketFile(filename, map)
```

or can be read from a file as

```
>>> (ierr, map2) = EpetraExt.MatrixMarketFileToBlockMap(filename, comm)
```

where ierr is the return error code. Analogously, Epetra.MultiVector and Epetra.CrsMatrix objects can be saved in a file as follows:

```
>>> EpetraExt.MultiVectorToMatrixMarketFile("x.mm", x)
>>> EpetraExt.RowMatrixToMatrixMarketFile("A.mm", A)
```

then read as

```
>>> (ierr, x2) = EpetraExt.MatrixMarketFileToMultiVector("x.mm", map)
>>> (ierr, A2) = EpetraExt.MatrixMarketFileToCrsMatrix("A.mm", map)
```

These functions are a powerful tool to exchange data between codes written in C++ using Trilinos and codes written in python using PyTrilinos.

See the EpetraExt documentation for more details.

5.4 TriUtils

This module, imported with the command

```
>>> from PyTrilinos import TriUtils
```

allows the creation of several matrices, in a way that mimics the MATLAB gallery function, and it can be useful for examples and testing.

Several matrices can be easily generated using module TriUtils. For details, we refer to the Trilinos tutorial (Chapter 5). Here, we just show how to generate a matrix corresponding to a 3D Laplacian on a structured Cartesian grid. Let nx, ny, nz be the number of nodes along the x-, y- and z-axes, respectively, and comm be the communicator previously created (see the Epetra module). Then, we can simply write:

```
>>> nx = 100
>>> ny = 100
>>> nz = 100
>>> gallery = TriUtils.CrsMatrixGallery("laplace_3d", comm)
>>> gallery.Set("nx", nx)
>>> gallery.Set("ny", ny)
>>> gallery.Set("nz", ny)
```

The linear system matrix, solution and right-hand side are obtained as

```
>>> A = gallery.GetMatrix()
>>> x = gallery.GetStartingSolution()
>>> b = gallery.GetRHS()
```

These objects are automatically destroyed when the `gallery` object is deleted. See the TriUtils documentation for more details. Note also that there is also a more advanced gallery of test and example problems available in the `PyTrilinos.Galeri` module.

5.5 Amesos

All `Amesos` objects are constructed from the function class `Amesos`. The main goal of this class is to allow the user to select any supported and enabled direct solver, simply by changing an input parameter. Let us suppose that `Amesos` has been configured and compiled with support for `SuperLU`. To solve a linear system with `SuperLU`, we first need to create a Solver object,

```
>>> from PyTrilinos import Amesos, Epetra
>>> factory = Amesos.Factory()
>>> solver = factory.Create("Superlu", problem)
```

Then, we can perform the symbolic and numeric factorizations using methods

```
>>> solver.SymbolicFactorization()
>>> solver.NumericFactorization()
```

The numeric factorization phase will check whether a symbolic factorization exists or not. If not, method `SymbolicFactorization()` is invoked. Solution is computed using

```
>>> solver.Solve()
```

The solution phase will check whether a numeric factorization exists or not. If not, method `NumericFactorization()` is called. Users must provide the nonzero structure of the matrix for the symbolic phase, and the actual nonzero values for the numeric factorization. Right-hand side and solution vectors must be set before the solution phase.

Note that using the `Amesos` module the user can use within Python the following packages: `KLU`, `LAPACK`, `UMFPACK`, `SuperLU`, `SuperLU_DIST`, `TAUCS`, `PARDISO`, `DSCPACK`, `MUMPS`, `DSCPACK`. See the Amesos documentation for more details.

5.6 AztecOO

Often, large sparse and distributed linear systems are solved using iterative solvers of Krylov type, like for example conjugate gradient or GMRES. Within `PyTrilinos`, iterative solvers are accessed via the `AztecOO` module. As an example, let us consider the set of instructions required to adopt a non-preconditioned CG, with 1550 maximum iterations and a tolerance of 10ˆ(-5) on the relative residual:

```
>>> from PyTrilinos import AztecOO
>>> solver = AztecOO.AztecOO(A, x, b)
>>> solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.AZ_cg)
>>> solver.SetAztecOption(AztecOO.AZ_precond, AztecOO.AZ_none)
>>> solver.Iterate(1550, 1e-5)
```

Non-preconditioned methods rarely converge. `AztecOO` offers one-level overlapping domain decomposition preconditioner (with exact and inexact subdomain solvers) and multilevel preconditioners (see the ML module overview). The first can be specified as follows:

```
>>> solver.SetAztecOption(AztecOO.AZ_precond, AztecOO.AZ_dom_decomp)
>>> solver.SetAztecOption(AztecOO.AZ_subdomain_solve, AztecOO.AZ_ilu)
>>> solver.SetAztecOption(AztecOO.AZ_overalp, 1)
>>> solver.SetAztecOption(AztecOO.AZ_graph_fill, 1)
```

For more details on the available parameters, see the AztecOO documentation. Note also that `Teuchos.ParameterList` is now supported by `AztecOO`, so you can set the previous options equivalently with

```
>>> solver.SetParameters({"precond": "dom_decomp",
...                       "subdomain_solve": "ilu",
...                       "overlap": 1,
...                       "graph_fill": 1})
```

5.7 ML

To define a multilevel preconditioner as defined by the `ML` package, we first have to set up the required parameters in a python dictionary. A list of supported parameter can be found in the `ML` user's guide. Here, we specify 3 maximum levels, verbose output (10), and symmetric Gauss-Seidel smoother. Aggregates are computed using the Uncoupled scheme.

```
>>> from PyTrilinos import ML
>>> mlList = {
...  "max levels"        : 3,
...  "output"            : 10,
...  "smoother: type"    : "symmetric Gauss-Seidel",
...  "aggregation: type" : "Uncoupled"
... }
```

Then, we create the preconditioner and compute it,

```
>>> prec = ML.MultiLevelPreconditioner(A, False)
>>> prec.SetParameterList(mlList)
>>> prec.ComputePreconditioner()
```

Finally, we set up the solver, specifying `prec` as the preconditioner:

```
>>> solver = AztecOO.AztecOO(A, x, b)
>>> solver.SetPrecOperator(prec)
>>> solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.AZ_cg)
>>> solver.SetAztecOption(AztecOO.AZ_output, 16)
>>> solver.Iterate(1550, 1e-5)
```

Please check the ML documentation for more details.

6 PyTrilinos.Teuchos

The C++ version of the `Teuchos` package provides a large variety of tools and utilities. Many of these tools and utilities are already supported in python by standard library modules. As a result, much of `Teuchos`, such as the command-line interpreter, has not been given a python interface.

What has been wrapped centers around the `ParameterList` class, an important utility class that is used by several Trilinos packages for communicating arbitrary-type parameters between users and packages. The classes `XMLObject`, `XMLParameterListWriter` and `XMLParameterListReader` are used to convert between XML and `ParameterList` objects. The classes `XMLInputSource`, `FileInputSource`, and `StringInputSource` can be used to convert text to XML objects. The `Teuchos` communicator classes, `Comm`, `SerialComm`, `MpiComm` and `DefaultComm` are also wrapped.

Often, the `Teuchos` module is invisible to the user. The `ParameterList` class is analagous to the python dictionary (with the restriction that the dictionary keys must be strings), and python programmers can provide a python dictionary wherever a `ParameterList` is expected. `Teuchos` is imported by the package that uses the `ParameterLists` and converts between dictionaries and `ParameterLists` automatically.

The user can create a `Teuchos.ParameterList` directly, using the constructor, `set` and `sublist` methods, if he so chooses, and methods that accept `ParameterLists` will work as expected. It is really just a question of verbosity and elegance that argues in favor of using a python dictionary.

The python implementation of the `ParameterList` class has been expanded extensively. Its constructor can accept a python dictionary, and several methods and operators have been added to the class so that it behaves somewhat like a dictionary.

C++ `ParameterLists` are designed to support parameters of arbitrary type. The python implementation supports a subset of types *a priori* :

Python type	Dir	C/C++ type
`bool`	<->	`bool`
`int`	<->	`int`
`float`	<->	`double`
`str`	<--	`char *`
`str`	<->	`std::string`
`dict`	-->	`ParameterList`
`ParameterList`	<->	`ParameterList`

The C++ `ParameterList` class supports `begin()` and `end()` methods for iterating over the parameters. These methods are disabled in the python implementation, in favor of the dictionary iterator methods: `__iter__()`, `iteritems()`, `iterkeys()` and `itervalues()`.

6.1 ParameterList

The `ParameterList` class is augmented to behave somewhat like a python dictionary. Here are the following differences between the C++ and python implementations:

- Ignored methods: `begin()`, `end()`, `entry()`, `getEntryPtr()`, `getPtr()`, `isType()`, `name(ConstIterator)` and `setEntry()`.
- Dictionary constructors. A `ParameterList` can be constructed using a dictionary:

  ```
  plist = Teuchos.ParameterList(dict[,string])
  ```

  where `dict` is a dictionary whose keys are all strings and whose values are of types supported in the above table. The string name argument is optional and defaults to `"ANONYMOUS"`.
- Set methods. The templated C++ `set()` method is replaced in python with a method that takes a string name and a python object of supported type. For example:

  ```
  plist = Teuchos.ParameterList()
  plist.set("b",True)
  plist.set("i",10)
  plist.set("f",2.718)
  plist.set("s","Trilinos")
  plist.set("d",{"a":1, "b":2})
  ```

- Get methods. The templated C++ `get()` method is replaced in python with a method that returns a python object. From the previous example:

  ```
  print plist.get("f")
  print plist.get("d")
  ```

  will output:

  ```
  2.718
  {'a': 1, 'b': 2}
  ```

- The `setParameters()` method can take either a `ParameterList` or a python dictionary as its argument. The `ParameterList` is updated to contain all of the entries of the argument.

- Printing. Since `print` is a python keyword, the `print()` C++ method has been renamed `_print` in python. It takes an optional file argument that defaults to standard output. Its output is the same as the C++ implementation.

  The `__str__()` method returns a string representation of the `ParameterList` as though it were a python dictionary. The python `eval` function applied to the output of `__str__()` will produce an equivalent dictionary.

  The `__repr__()` method returns the `__str__()` output encapsulated by `ParameterList(...)`. The python `eval` function applied to the output of `__repr__()` will produce an equivalent `ParameterList`.

- The `unused()` method in python takes an optional python file object as its argument, defaulting to standard output.

- Parameter type determination. With the templated `isType()` methods disabled, type determination of python `ParameterList` entries is accomplished with the `type()` method, which returns python type objects. From our previous example:

  ```
  print plist.type("b")
  print plist.type("s")
  ```

  results in:

  ```
  <type 'bool'>
  <type 'str'>
  ```

  A non-existent key given as the argument will raise a `KeyError` exception.

- An `asDict()` method has been added that returns the contents of the `ParameterList` converted to a python dictionary.

The following python dictionary methods and operators are added to the python implementation of the `ParameterList` class:

- Comparison. The `==` and `!=` comparison operators work for `ParameterList` objects. Comparison against other `ParameterList` objects, as well as python dictionaries, is supported. Less-than and greater-than operators also work, mimicking the default dictionary behavior, but are less useful.

- The `in` operator also works, searching the parameter names:

  ```
  print "a" in plist
  print "b" in plist
  ```

  produces:

```
False
True
```

- Length method. The python `len()` function works on `ParameterLists`:

  ```
  print len(plist)
  ```

  gives:

  ```
  5
  ```

- Iteration. To iterate over the parameters in a `ParameterList`, treat it like a dictionary:

  ```
  for key in plist:
    print key, ":", plist[key]
  ```

  will result in the output:

  ```
  b : True
  d : {'a': 1, 'b': 2}
  f : 2.718
  i : 10
  s : Trilinos
  ```

  Note that the order of the parameters is somewhat indeterminant, as with dictionaries, because the iteration object is obtained from an equivalent dictionary, and dictionaries are ordered by hash function.

- Index notation. Like dictionaries, parameters can be set and gotten using square brackets:

  ```
  plist["zero"] = 0
  e = plist["f"]
  ```

- Update methods. An `update()` method has been added that can accept either a `ParameterList` or a python dictionary. Otherwise, it behaves just as the dictionary method, which is functionally equivalent to the `setParameters()` method.

- Other new methods for the python implementation that behave just as with dictionaries: `has_key()`, `items()`, `iteritems()`, `iterkeys()`, `itervalues()`, `keys()` and `values()`.

Note that the C++ implementation of the `ParameterList` class does not support parameter deletion. Therefore, python dictionary methods that delete items, such as `pop()` or `__delitem__()`, have not been added to the `ParameterList` class.

6.2 XML Support

`PyTrilinos.Teuchos` supports several classes related to XML. They are:

- `XMLObject` - The python implementation of this class is largely unchanged from the C++ interface, with the following exceptions:
  - The constructor that takes an `XMLObjectImplem*` argument has been removed. The `XMLObjectImplem` class is hidden from the python user.
  - A `__str__()` method has been added, so that it is possible to `print` an `XMLObject` object. It returns the same string as the `toString()` method, but if `toString()` raises an exception (such as when the `XMLObject` is empty), the `__str__()` method returns the empty string.

- `XMLParameterListWriter` - This simple class has been implemented unchanged from its C++ interface. Its purpose is to convert `ParameterList` objects into `XMLObject` objects.

- XMLParameterListReader - This simple class has been implemented unchanged from its C++ interface. Its purpose is to convert XMLObject objects into ParameterList objects.

- XMLInputSource - This is a base class from which FileInputSource and StringInputSource derive. The source() method is ignored.

- FileInputSource - This is a class for reading an XML object from a file. The source() method is ignored.

- StringInputSource - This is a class for reading an XML object from a string. The source() method is ignored.

7 PyTrilinos.Epetra

The Trilinos Epetra package, for historical portability reasons, does not use namespaces. Instead, "Epetra_" is prepended to every class and function name. The python implementation, however, does utilize the Epetra namespace, and the "Epetra_" prefix has been stripped from all of the class and function names. Therefore, Epetra_Object in C++ is implemented as Epetra.Object in python.

Epetra supports a large number of classes. They can be categorized as follows:

- Fundamental Classes
- Communicators
- Maps
- Vectors
- SerialDense Classes
- Graphs
- Operators

7.1 Fundamental Classes

- Epetra.Object: this is the base class for the majority of Epetra classes. In C++, it supports a Print() method that takes an output stream as an argument. In the python implementation for this and all derived classes, this method takes an optional file object argument whose default value is standard out.

  A __str__() method has also been added to the Epetra.Object class that returns the results of Print() in a string, so that the print command will work on Epetra objects. The Print() methods are designed to run correctly in parallel, so do not execute print on an Epetra object conditionally on the processor number. For example, do not do

  ```
  if comm.MyPID() == 0:  print epetra_obj
  ```

  or it will hang your code.

- Epetra.Util: The Sort() method is not supported.

7.2 Communicators

If PyTrilinos was compiled with MPI support, then MPI_Init() will be called internally when the Epetra module is imported (the Epetra module will also arrange for MPI_Finalize() to be called upon termination of the python interpreter).

Parallelism in Trilinos (and PyTrilinos) is largely encapsulated in the Epetra communicator classes. In addition to the base class Epetra.Comm and the derived Epetra.SerialComm and Epetra.MpiComm classes, the Epetra.PyComm() function has been added to the python implementation that returns the appropriate communicator type, depending on whether or not PyTrilinos was compiled with MPI support.

The python interfaces for various communicator methods are slightly different than the C++ interfaces. If comm is an Epetra communicator, then the following methods have the given interfaces:

- `comm.Broadcast(numpy.ndarray obj, int root)`
- `comm.GatherAll(PyObject obj) -> numpy.ndarray`
- `comm.SumAll(PyObject obj) -> numpy.ndarray`
- `comm.MaxAll(PyObject obj) -> numpy.ndarray`
- `comm.MinAll(PyObject obj) -> numpy.ndarray`
- `comm.ScanSum(PyObject obj) -> numpy.ndarray`

In the Broadcast method, the numpy.ndarray data from the root processor is broadcast in-place to all of the other processors. Arrays of ints, longs, doubles and now strings are supported. In the remaining methods, the PyObject obj input argument must be a python object that can be converted to an integer, long, or double numpy.ndarray, and the return value is a numpy.ndarray of the same type and size. In C++, these routines have integer error return codes. In python, a non-zero return code is converted to an exception.

7.3 Maps

Epetra maps describe the distribution of global vector indexes across processors. The python interface for the following classes is slightly modified from the C++ implementations:

- Epetra.BlockMap is the (concrete) base class for other Epetra maps. It supports a given number of global elements, where each element can support a (possibly variable) number of data points. But it is the elements that are distributed among the processors. The following constructors are supported:
  - `BlockMap(numGE, elSize, iBase, comm)`
  - `BlockMap(numGE, numME, elSize, iBase, comm)`
  - `BlockMap(numGE, myGEs, elSize, iBase, comm)`
  - `BlockMap(numGE, myGEs, elSizes, iBase, comm)`
  - `BlockMap(map)`

  where comm is an Epetra communicator; numGE is the integer number of global elements; numME is the integer number of elements on this processor; elSize is the integer element size; iBase is the integer index base (typically 0 or 1); myGEs is a sequence of integers representing the global indexes on this processor; elSizes is a sequence of integers representing the number of data points for each element on this processor; and map is a BlockMap.

  Instead of two C++ RemoteIDList methods, there is only one with the following interface:

– `BlockMap.RemoteIDLiSt(GIDList) -> (PIDList, LIDList, sizeList)`

where `GIDList` is a sequence of integer global IDs, and `PIDList`, `LIDList` and `sizeList` are `numpy.ndarray` objects of integers representing the processor IDs, local IDs and element sizes, respectively. Other `BlockMap` methods are altered to have the following python interfaces:

– `BlockMap.FindLocalElementID(pointID) -> (elementID, elementOffset)`
– `BlockMap.MyGlobalElements() -> numpy.ndarray`
– `BlockMap.FirstPointInElementList() -> numpy.ndarray`
– `BlockMap.ElementSizeList() -> numpy.ndarray`
– `BlockMap.PointToElementList() -> numpy.ndarray`

- `Epetra.Map` is a simpler form of `BlockMap`, in which the size of each element is restricted to 1. The python implementation supports the following constructors:

  – `Map(numGE, iBase, comm)`
  – `Map(numGE, numME, iBase, comm)`
  – `Map(numGE, myGEs, iBase, comm)`
  – `Map(map)`

  where `comm` is an Epetra communicator; `numGE` is the integer number of global elements; `numME` is the integer number of local elements on this processor; `iBase` is the integer index base (typically 0 or 1); `myGEs` is a sequence of integer global indexes on this processor; and `map` is an `Epetra.Map`

- `Epetra.Export` The `Export` class has the following altered python method interfaces:

  – `Export.PermuteFromLIDs() -> numpy.ndarray`
  – `Export.PermuteToLIDs() -> numpy.ndarray`
  – `Export.RemoteLIDs() -> numpy.ndarray`
  – `Export.ExportLIDs() -> numpy.ndarray`
  – `Export.ExportPIDs() -> numpy.ndarray`

- `Epetra.Import` The `Import` class has the following altered python method interfaces:

  – `Import.PermuteFromLIDs() -> numpy.ndarray`
  – `Import.PermuteToLIDs() -> numpy.ndarray`
  – `Import.RemoteLIDs() -> numpy.ndarray`
  – `Import.ExportLIDs() -> numpy.ndarray`
  – `Import.ExportPIDs() -> numpy.ndarray`

7.4 Vectors

One of the most fundamental data structures for scientific computing is the contiguous array of homogeneous scalars. For scientific python, this data structure has historically been provided by the `Numeric` module. This was followed by `numarray`, which added capabilities `Numeric` lacked, but never replicated all of the `Numeric` module's functionality. These two modules have now been replaced by the `numpy` module, whose purpose is to provide a single multidimensional array module to python, acceptable by all factions of the scientific python community.

In `Epetra`, the contiguous array data structure is provided by several vector classes. In general, these classes lack many of the bells and whistles of `numpy.ndarray` objects, but they provide at least one unique feature: extensive parallel support. `Epetra` vectors are assumed to be distributed over one or more processors, as specified by a map object.

To give PyTrilinos users maximum flexibility, the python implementations of the `Epetra` vector classes `Epetra.Vector`, `Epetra.MultiVector`, `Epetra.FEVector` and `Epetra.IntVector` inherit from the `numpy.lib.user_array.container` class. These classes are thus instances of both `Epetra` vectors and `numpy.ndarray`. The `Epetra` `Print()` method provides `Epetra` output, and the `__str__()` method provides `numpy` output. The key to this approach is providing internal constructors that create an `Epetra` vector and a `numpy.ndarray` that both point to the same data buffer. If you extract a slice from an `Epetra.Vector` or `Epetra.MultiVector` object, the result is a new `Epetra.[Multi]Vector` with a new `Epetra.Map` that reflects the global IDs of the sliced array.

The new constructors, as well as other methods with alternate python interfaces, are described below:

- `Epetra.Vector` provides an array of double precision data. `PyTrilinos` packages will always interpret an `Epetra.Vector` as a 1D array, but the python implementation allows any shape that is legal for the given length of the vector. Its constructors and methods with new interfaces are
  - `Vector(BlockMap map, bool zeroOut=True)`

    Create a new `Vector` with a size and distribution defined by `map`. Initialize to zero unless `zeroOut` is False.
  - `Vector(BlockMap map, PyObject array)`

    Create a `Vector` that uses a PyObject to initialize the data. The `array` object can be either a `numpy.ndarray` or any sequence that can be used to construct a `numpy.ndarray`, which then provides the data buffer. The length of the array object on each processor must match the number of elements on each processor specified by the `map`. If not, an exception is raised.
  - `Vector(DataAccess CV, MultiVector source, int index)`

    Create a `Vector` by extracting a single vector from a `MultiVector`, specified by `index`. `CV` should be either `Epetra.Copy` or `Epetra.View`.
  - `Vector(PyObject array)`

    Convert `array` to a `numpy.ndarray` if necessary and use this to provide the data buffer for a new `Vector`. The underlying communicator is `Epetra.SerialComm`, used as the basis for a simple map.
  - `Vector(Vector source)`

    Copy constructor.
  - `ExtractCopy() -> numpy.ndarray`
  - `ExtractView() -> numpy.ndarray`
  - `Dot(Vector A) -> double`
  - `Norm1() -> double`
  - `Norm2() -> double`
  - `NormInf() -> double`
  - `NormWeighted(Vector weights) -> double`
  - `MinValue() -> double`
  - `MaxValue() -> double`
  - `MeanValue() -> double`
  - `ReplaceGlobalValues(PyObject values, PyObject indices) -> int`
  - `ReplaceGlobalValues(int blockOffset, PyObject values, PyObject indices) -> int`
  - `ReplaceMyValues(PyObject values, PyObject indices) -> int`

- ReplaceMyValues(int blockOffset, PyObject values, PyObject indices) -> int
- SumIntoGlobalValues(PyObject values, PyObject indices) -> int
- SumIntoGlobalValues(int blockOffset, PyObject values, PyObject indices) -> int
- SumIntoMyValues(PyObject values, PyObject indices) -> int
- SumIntoMyValues(int blockOffset, PyObject values, PyObject indices) -> int

- `Epetra.MultiVector` is actually the base class for `Vector`. PyTrilinos packages will always interpret a `MultiVector` as a 2D array, but the python implementation allows them to have two or more dimensions. Thus the `shape` attribute can be changed to a tuple of integers of length at least two whose elements' product is the total size of the array.
  - `MultiVector(BlockMap map, int n, bool zeroOut=True)`

    Create a new `MultiVector` with `n` vectors, with length and distribution according to `map`. Initialize to zero unless `zeroOut` is False.
  - `MultiVector(BlockMap map, PyObject array)`

    Create a `MultiVector` that uses a PyObject to initialize the data. The `array` object is converted to a `numpy.ndarray` if necessary, which then provides the data buffer. If the `numpy.ndarray` is one-dimensional, it is upcast to 2D with a first dimension of one. The product of the second and any subsequent dimensions of the array object on each processor must match the number of elements on each processor specified by the `map`. If not, an exception is raised.
  - `MultiVector(DataAccess CV, MultiVector source, PyObject range)`

    Create a `MultiVector` by extracting a subset of vectors from a `MultiVector`. The `range` object should evaluate to a sequence of integers that specify the subset. `CV` should be either `Epetra.Copy` or `Epetra.View`.
  - `MultiVector(PyObject array)`

    Convert `array` to a `numpy.ndarray` if necessary and use this to provide the data buffer for a new `MultiVector`. The first dimension of the `numpy.ndarray` specifies the number of vectors. If the `numpy.ndarray` is one-dimensional, it will be upcast to a 2D array with a first dimension of one. The underlying communicator is `Epetra.SerialComm`, used as the basis for a simple map.
  - `MultiVector(MultiVector source)`

    Copy constructor.
  - `ExtractCopy() -> numpy.ndarray`
  - `ExtractView() -> numpy.ndarray`
  - `Dot(MultiVector a) -> numpy.ndarray`
  - `Norm1() -> numpy.ndarray`
  - `Norm2() -> numpy.ndarray`
  - `NormInf() -> numpy.ndarray`
  - `NormWeighted(MultiVector weights) -> numpy.ndarray`
  - `MinValue() -> numpy.ndarray`
  - `MaxValue() -> numpy.ndarray`
  - `MeanValue() -> numpy.ndarray`

- `Epetra.FEVector` derives from `Epetra.MultiVector` and provides some methods that are convenient for finite element vector assembly. The C++ version recently implemented the multivector nature of the FEVector. The python wrappers were updated to reflect this.

- `Epetra.IntVector` provides an array of integers. PyTrilinos packages will always interpret an `Epetra.IntVector` as a 1D array, but the python implementation allows any shape that is legal for the given length of the vector.
  - `IntVector(BlockMap map, bool zeroOut=True)`
    Create a new `IntVector` with a size and distribution defined by `map`. Initialize to zero unless `zeroOut` is False.
  - `IntVector(BlockMap map, PyObject array)`
    Create an `IntVector` that uses a PyObject to initialize the data. The `array` object is converted to a `numpy.ndarray` if necessary, which then provides the data buffer. The length of the array object on each processor must match the number of elements on each processor specified by the `map`. If not, an exception is raised.
  - `IntVector(PyObject array)`
    Convert `array` to a `numpy.ndarray` and use this to provide the data buffer for a new `IntVector`. The underlying communicator is `Epetra.SerialComm`, used as the basis for a simple map.
  - `IntVector(IntVector source)`
    Copy constructor.
  - `ExtractCopy() -> numpy.ndarray`
  - `ExtractView() -> numpy.ndarray`
  - `Values() -> numpy.ndarray`

7.5 SerialDense Classes

As with `Epetra` vector objects, several of the `Epetra` SerialDense objects multiply inherit from the `numpy.lib.user_array.container` class. These classes are:

- `IntSerialDenseMatrix`
- `IntSerialDenseVector`
- `SerialDenseMatrix`
- `SerialDenseVector`

The following SerialDense class methods have an altered calling signature because the python version is smart enough to determine the dimensions automatically:

- `SerialDenseSolver.IPIV() -> numpy.ndarray`
- `SerialDenseSolver.A() -> numpy.ndarray`
- `SerialDenseSolver.B() -> numpy.ndarray`
- `SerialDenseSolver.X() -> numpy.ndarray`
- `SerialDenseSolver.AF() -> numpy.ndarray`
- `SerialDenseSolver.FERR() -> numpy.ndarray`
- `SerialDenseSolver.BERR() -> numpy.ndarray`
- `SerialDenseSolver.R() -> numpy.ndarray`
- `SerialDenseSolver.C() -> numpy.ndarray`

The `SerialDenseSolver.ReciprocalConditionEstimate()` method takes no arguments, returns the double precision result and raises an exception if the underlying C++ int result is nonzero.

7.6 Graphs

The `CrsGraph` class has two new constructors:

- `CrsGraph(DataAccess CV, BlockMap rowMap, PyObject numIndicesList, bool staticProfile=False)`
- `CrsGraph(DataAccess CV, BlockMap rowMap, BlockMap colMap, PyObject numIndicesList, bool staticProfile=False)`

The argument `CV` is either `Epetra.Copy` or `Epetra.View`, `rowMap` and `colMap` are maps that describe the domain decomposition of global row indices and column indices respectively, `numIndicesList` is a python sequence of integers that lists the number of non-zeros for each row, and `staticProfile` is a boolean that flags whether the number of indices per row is exact or approximate.

The following `CrsGraph` methods have simplified argument lists:

- `CrsGraph.ExtractGlobalRowCopy(int globalRow) -> numpy.ndarray`
- `CrsGraph.ExtractMyRowCopy(int myRow) -> numpy.ndarray`
- `CrsGraph.InsertGlobalIndices(int globalRow, PyObject indices) -> int`
- `CrsGraph.InsertMyIndices(int myRow, PyObject indices) -> int`
- `CrsGraph.RemoveGlobalIndices(int globalRow, PyObject indices) -> int`
- `CrsGraph.RemoveMyIndices(int myRow, PyObject indices) -> int`

The following `CrsGraph` methods do not have python wrappers:

- `CrsGraph.ExtractGlobalRowView()`
- `CrsGraph.ExtractMyRowView()`

7.7 Operators

Perhaps the simplest `Epetra` operators to create and use are the `CrsMatrix` and `VbrMatrix`. You simply call their constructors in the normal way, populate them, and then call their `FillComplete()` methods. These classes have two new constructors:

- `CrsMatrix(DataAccess CV, Map rowMap, PyObject numIndicesList, bool staticProfile=False)`
- `CrsMatrix(DataAccess CV, Map rowMap, Map colMap, PyObject numIndicesList, bool staticProfile=Fals`

and

- `VbrMatrix(DataAccess CV, Map rowMap, PyObject numBlockEntriesPerRow)`
- `VbrMatrix(DataAccess CV, Map rowMap, Map colMap, PyObject numBlockEntriesPerRow)`

The argument `CV` is either `Epetra.Copy` or `Epetra.View`, `rowMap` and `colMap` are maps that describe the domain decomposition of global row indices and column indices respectively, `numIndicesList` is a python sequence of integers that lists the number of non-zeros for each row, and `staticProfile` is a boolean that flags whether the number of indices per row is exact or approximate. For the `VbrMatrix` constructors, `numBlockEntriesPerRow` can be either an interger, constant for all rows, or a sequence with as many entries as the matrix has rows.

The following `CrsMatrix` methods have simplified argument lists:

- `CrsMatrix.ExtractGlobalRowCopy(int globalRow) -> (numpy.ndarray,numpy.ndarray)`

- CrsMatrix.ExtractMyRowCopy(int myRow) -> (numpy.ndarray,numpy.ndarray)
- CrsMatrix.InsertGlobalValues(int globalRow, PyObject indices, PyObject values) -> int
- CrsMatrix.InsertMyValues(int myRow, PyObject indices, PyObject values) -> int
- CrsMatrix.RemoveGlobalValues(int globalRow, PyObject indices, PyObject values) -> int
- CrsMatrix.RemoveMyValues(int myRow, PyObject indices, PyObject values) -> int
- CrsMatrix.SumIntoGlobalValues(int globalRow, PyObject indices, PyObject values) -> int
- CrsMatrix.SumIntoMyValues(int myRow, PyObject indices, PyObject values) -> int

The following CrsMatrix methods do not have python wrappers:

- CrsMatrix.ExtractGlobalRowView()
- CrsMatrix.ExtractMyRowView()

The CrsMatrix class also has indexing enabled. Thus, if A is an CrsMatrix, matrix elements can be assigned with the syntax

```
>>> A[i,j] = 2.7182818284590451
```

Under certain conditions, index notation can be used to retrieve single matrix elements (with a row and column index), or all of the elements in the row (with a single row index). For single matrix elements, a column map must exist, which can be provided in the constructor, or computed automatically when FillComplete() is called. To extract row data with a single index, FillComplete() is required to have been called.

```
>>> comm = Epetra.PyComm()
>>> map = Epetra.Map(9,0,comm)
>>> A = Epetra.CrsMatrix(Epetra.Copy,map,3)
>>> for gid in map.MyGlobalElements():
...     lid = map.LID(gid)
...     if gid in (0,8): A.InsertGlobalIndices(lid,[1],[gid])
...     else: A.InsertGlobalIndices(lid,[-1,2,-1],[gid-1,gid,gid+1])
...
>>> A.FillComplete()
>>> print A[8,8]
1.0
>>> print A[4]
[ 0.  0.  0. -1.  2. -1.  0.  0.  0.]
```

A more flexible way of defining Epetra operators is to define your own classes by inheriting from pure virtual Epetra operator classes. The following classes can be sub-classed successfully in python. That is, you can define a python class that inherits from one of these classes, define the appropriate methods, such as Apply(), and the infrastructure is in place such that C++ solver routines (such as the AztecOO.Iterate() method) can call back to your python method successfully.

- Operator
- InvOperator

- `RowMatrix`
- `BasicRowMatrix`

Python classes derived from these callback-supporting base classes ("directors" in the parlance of SWIG, which is the tool that generates the wrapper code) must call their base class `__init__()` method:

```
from PyTrilinos import Epetra

class MyOperator(Epetra.Operator):
    def __init__(self):
        Epetra.Operator.__init__(self)
        self.__label = "MyOperator"
```

At a bare minimum, you must define a `Label()` method and an `Apply()` method that support the argument prototypes found in the `Epetra` documentation:

```
def Label(self):
    return self.__label
def Apply(self, x, y):
    try:
        y[:,0   ] = x[:,0]
        y[:,1:-1] = 2*x[:,1:-1] - x[:,:-2] - x[:,2:]
        y[:,-1  ] = x[:,-1]
        return 0
    except Exception, e:
        print "A python exception was raised in MyOperator.Apply:"
        print e
        return -1
```

A few notes about the `Apply()` method:

- Arguments `x` and `y` will be sent by a C++ solver (such as `AztecOO`) with C++ type `Epetra_MultiVector`. Before they are passed along to your python method, they will be converted to the `numpy`-hybrid type `Epetra.MultiVector`. Hence we are able, within the method, to access slices of `x` and `y`, which is a very efficient way to calculate with buffers of data.
- Also, because `x` and `y` are `MultiVector` objects, they should be treated as 2-dimensional, with the first dimension representing the individual vectors. As in this example, this usually just means making sure the first index is a colon.
- The calculations here are done within a `try` block. This is because if your `Apply()` accidentally raises an exception, the callback mechanism isn't currently able to pass the error message to you. Hence, we catch all exceptions (base class `Exception`), print the error message, and return a non-zero error code to tell the calling function that we failed. This technique is recommended for all callback functions of any complexity. It adds almost no overhead and is a great help for debugging.
- In this particular example, we have taken advantage of python's negative indexing (which indexes from the end of a sequence). For this reason, this operator works on different size vectors. If the input vectors, `x` and `y`, have different shapes, then the assignment statements will raise an exception, which we will catch, print, and then inform the calling routine by returning -1.

The operative inheritance chain here is that `SrcDistObject` and `Operator` are both base classes for `RowMatrix`, which is a base class for `BasicRowMap`. Each of these classes has virtual methods you may want to implement with the given input prototypes and output argument:

- `SrcDistObject`
  - `Map() -> BlockMap`
- `Operator`
  - `SetUseTranspose(bool useTranspose)`
  - `UseTranspose() -> bool`
  - `Apply(MultiVector x, MultiVector y) -> int`
  - `ApplyInverse(MultiVector x, MultiVector y) -> int`
  - `HasNormInf() -> bool`
  - `NormInf() -> double`
  - `Label() -> string`
  - `Comm() -> Comm`
  - `OperatorDomainMap() -> Map`
  - `OperatorRangeMap() -> Map`
- `RowMatrix`
  - `NumMyRowEntries(int myRow, numpy.ndarray numEntries) -> int`
  - `MaxNumEntries() -> int`
  - `ExtractMyRowCopy(int myRow, int length, numpy.ndarray numEntries, numpy.ndarray values, numpy.ndarray indices) -> int`
  - `ExtractDiagonalCopy(Vector diagonal) -> int`
  - `Multiply(bool useTranspose, MultiVector x, MultiVector y) -> int`
  - `Solve((bool upper, bool trans, bool unitDiagonal, MultiVector x, MultiVector y) -> int`
  - `InvRowSum(Vector x) -> int`
  - `LeftScale(Vector x) -> int`
  - `InvColSums(Vector x) -> int`
  - `RightScale(Vector x) -> int`
  - `Filled() -> bool`
  - `NormOne() -> double`
  - `NumGlobalNonzeros() -> int`
  - `NumGlobalRows() -> int`
  - `NumGlobalCols() -> int`
  - `NumGlobalDiagonals() -> int`
  - `NumMyNonzeros() -> int`
  - `NumMyRows() -> int`
  - `NumMyCols() -> int`
  - `NumMyDiagonals() -> int`
  - `LowerTriangular() -> bool`
  - `UpperTriangular() -> bool`
  - `RowMatrixRowMap() -> Map`
  - `RowMatrixColMap() -> Map`
  - `RowMatrixImporter() -> Import`

Of particular importance to the python interface of the `RowMatrix` and `BasicRowMatrix` classes are the `NumMyRowEntries` and `ExtractMyRowCopy` methods, because these method's C++ argument lists have output arguments: `int &NumEntries`, `double *Values`, and `int *Indices`. To convert these into mutable python objects that can properly act as output arguments, these arguments are converted to `numpy.ndarray` objects whose data buffers point to the passed-in C++ data. It is natural to access the `values` and `indices` as arrays with bracket notation for setting values, but the `numEntries` argument, which is an array of length 1, must also be accessed this way:

```
def ExtractMyRowCopy(self, myRow, length, numEntries, values, indices):
    globalRow = self.RowMatrixRowMap().GID(myRow)
    if globalRow == -1:
        return -1
    if globalRow == 0 or globalRow == self.NumGlobalRows()-1:
        if (length < 1):
            return -2
        numEntries[0] = 1
        values[0]     = 1.0
        indices[0]    = myRow
    else:
        if (length < 3):
            return -2
        numEntries[0] = 3
        values[:3]    = [   -1.0,    2.0,    -1.0]
        indices[:3]   = [myRow-1, myRow, myRow+1]
    return 0
```

This example assumes you have properly implemented the `RowMatrixRowMap()` and `NumGlobalRows()` methods.

8 PyTrilinos.EpetraExt

The `EpetraExt` package supports a wide variety of extensons to the `Epetra` package, only a handful of which have currently been given python wrappers. These wrappers can be categorized as follows:

- Graph Coloring Classes
- Input Functions
- Output Functions
- Input/Output Classes
- Matrix-Matrix Functions
- Model Evaluator Classes

8.1 Graph Coloring Classes

`EpetraExt` has two classes for aiding with creating color maps.

- `CrsGraph_MapColoring`
- `CrsGraph_MapColoringIndex`

The first is the functor `CrsGraph_MapColoring`. Note than only the default constructor is currently supported. Once such an object is created, you can call it with an `Epetra.CrsGraph` argument to call a coloring algorithm and obtain an `Epetra.MapColoring` object.

The second class is the `CrsGraph_MapColoringIndex` functor. Use an `Epetra.MapColoring` object in the constructor and an `Epetra.CrsGraph` object as the argument when you call it.

8.2 Input Functions

- `MatlabFileToCrsMatrix(str filename, Epetra.Comm) -> Epetra.CrsMatrix`
- `MatrixMarketFileToBlockMap(str filename, Epetra.Comm) -> Epetra.BlockMap`
- `MatrixMarketFileToBlockMaps(str filename, Epetra.Comm) -> (Epetra.BlockMap rowMap, Epetra.BlockMap colMap, Epetra.BlockMap rangeMap, Epetra.BlockMap domainMap)`
- `MatrixMarketFileToCrsMatrix(str filename, Epetra.Map rowMap, Epetra.Map colMap=None, Epetra.Map rangeMap=None, Epetra.Map domainMap=None) -> Epetra.CrsMatrix`
- `MatrixMarketFileToMap(str filename, Epetra.Comm) -> Epetra.Map`
- `MatrixMarketFileToMultiVector(str filename, Epetra.BlockMap) -> Epetra.MultiVector`

8.3 Output Functions

- `BlockMapToHandle(file handle, Epetra.BlockMap map) -> int`
- `BlockMapToMatrixMarketFile(str filename, Epetra.BlockMap map, str mapName=None, str descr=None, bool writeHeader=True) -> int`
- `MultiVectorToHandle(file handle, Epetra.MultiVector) -> int`
- `MultiVectorToMatlabFile(str filename, Epetra.MultiVector) -> int`
- `MultiVectorToMatlabHandle(file handle, Epetra.MultiVector) -> int`
- `MultiVectorToMatrixMarketFile(str filename, Epetra.MultiVector) -> int`
- `MultiVectorToMatrixMarketHandle(file handle, Epetra.MuiltiVector) -> int`
- `RowMatrixToHandle(file handle, Epetra.RowMatrix) -> int`
- `RowMatrixToMatlabFile(str filename, Epetra.RowMatrix) -> int`
- `RowMatrixToMatrixMarketFile(str filename, Epetra.RowMatrix) -> int`

8.4 Input/Output Classes

- `HDF5`
- `XMLReader`
- `XMLWriter`

The `HDF5` and `XMLReader` classes both support overloaded `Read()` methods in C++, but these cannot be type-disambiguated in python. Therefore, the `Read()` methods are ignored and replaced with python versions that have the type name in the method. Therefore, if `obj` is an `HDF5` or `XMLReader` object, then, these methods are supported:

- `obj.ReadMap(str) -> Epetra.Map`
- `obj.ReadMultiVector(str) -> Epetra.MultiVector`
- `obj.ReadCrsGraph(str) -> Epetra.CrsGraph`
- `obj.ReadCrsMatrix(str) -> Epetra.CrsMatrix`

In addition, the `HDF5` objects support the following additional methods:

- `obj.ReadBlockMap(str) -> Epetra.BlockMap`
- `obj.ReadIntVector(str) -> Epetra.IntVector`

8.5 Matrix-Matrix Functions

So far, only addition and multiplication are supported:

- `Add(Epetra.CrsMatrix A, bool flag, float valA, Epetra.CrsMatrix B, float valB) -> int`

  Compute `B <- valA * A + valB * B`. If flag is `True`, use the transpose of `A`. `B` must either have the structure of `A+B` or not yet have `FillComplete()` called on it.

- `Multiply(Epetra.CrsMatrix A, bool transposeA, ‘‘ ‘‘Epetra.CrsMatrix B, bool transposeB, Epetra.CrsMatrix C) -> int`

  Compute `C <- A * B`, where `transposeA` and `transposeB` control the transposition of `A` and `B` respectively. `C` must have the structure of `A * B`, or not yet have `FillComplete()` called on it.

8.6 Model Evaluator Classes

- `InArgs`

  A class that defines and encapsulates the input arguments to the model.

- `Evaluation`

  A class that defines how derivatives are evaluated (exactly, approximately, or very approximately).

- `DerivativeSupport`

  A class that encapsulates the linearity, multivector status and transpose status of a Derivative.

- `DerivativeProperties`

  A class that encapsulates the linearity, rank, and adjoint support of a Derivative.

- `DerivativeMultiVector`

  A class that encapsulates a MultiVector, its orientation and its parameter indexes.

- `Derivative`

  A class that can represent a derivative object as either an operator or a vector.

- `OutArgs`

  A class that defines and encapsulates the output arguments from the model.

- `ModelEvaluator`

  The primary model evaluator class. It can be used to define its InArgs and OutArgs, as well as evaluate the model.

Note that in C++, the first seven classes listed above are nested within the `ModelEvaluator` class. This arrangements creates problems when attempting to wrap the outer class, so the nested classes have been pulled out instead.

9 PyTrilinos.TriUtils

The `TriUtils` package is a set of utilities for the `Trilinos` project and is especially useful for testing purposes. Likewise, the `TriUtils` module is used by several of the python test and example scripts.

The primary difference between the C++ and python implementations is the function

- `ReadHb2Epetra (string, Epetra.Comm) -> (Epetra.Map, Epetra.CrsMatrix, Epetra.Vector x, Epetra.Vector b, Epetra.Vector exact)`

in which the results of the function are returned in a tuple.

10 PyTrilinos.Galeri

`Galeri` functions behave in python essentially as they do in C++. The one exception is the `ReadHB` function,

- `ReadHB(str filename, Epetra.Comm comm) -> (Epetra.Map map, Epetra.CrsMatrix A, Epetra.Vector x, Epetra.Vector b, Epetra.Vector exact)`

that packs all of its output arguments into a tuple that becomes the return value.

11 PyTrilinos.Amesos

The `Amesos` module supports the following third-party solver packages, assuming you have them installed on your system and have configured `Trilinos` and `Amesos` to use them.

- `LAPACK`
- `KLU`
- `UMFPACK`
- `ScaLAPACK`
- `SuperLU`
- `SuperLUDist`
- `TAUCS`
- `Pardiso`
- `DSCPACK`
- `MUMPS`

The python interface for `Amesos` is essentially the same as the C++ interface. This means that you need an `Epetra.LinearProblem` class that contains your right-hand side and solution `Epetra.Vector` or `Epetra.MultiVector`, and the linear system `Epetra.RowMatrix`. Note that the `Amesos` module allows `FORTRAN90` codes such as `MUMPS` to be used interactively!

12 PyTrilinos.AztecOO

The python interface to `AztecOO` does not differ significantly from the C++ interface. Note that the AztecOO-Teuchos support, allowing the use of `Teuchos.ParameterList` objects to specify `AztecOO` options, is implemented for the python interface when both Teuchos and AztecOO are enabled. Note also that `PyTrilinos` performs automatic conversions between python dictionaries and `Teuchos::ParameterList` objects.

13 PyTrilinos.IFPACK

The python interface for `IFPACK` is essentially the same as the C++ interface.

14 PyTrilinos.ML

The most notable difference between ML and its Python module is in the construction of the preconditioner. Given an `Epetra.RowMatrix` object (say, `A`), first you need a set of parameters, specified in a Python dictionary:

```
mlList = {"max levels"        : 3,
          "output"            : 10,
          "smoother: type"    : "symmetric Gauss-Seidel",
          "aggregation: type" : "Uncoupled"
         }
```

All parameters are specified as in C++; please check the ML page for more details.

Then, you can create the preconditioner (derived from the `Epetra.Operator` class) as follows:

```
prec = ML.MultiLevelPreconditioner(A, False)
prec.SetParameterList(mlList)
prec.ComputePreconditioner()
```

Note that you first need to instantiate `prec` using `False`, then let `prec` parse the parameters contained in `mlList`, and finally build the preconditioner. Using `prec` as a preconditioner for `AztecOO` may be done as simply as:

```
solver = AztecOO.AztecOO(A, x, y)
solver.SetPrecOperator(prec)
solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.AZ_cg);
solver.SetAztecOption(AztecOO.AZ_output, 16);
err = solver.Iterate(1550, 1e-5)
```

15 PyTrilinos.NOX

Python versions of C++ methods that expect `Teuchos::ParameterList` arguments accept either `Teuchos.ParameterList` objects or python dictionaries.

Since python objects are already reference counted, the `Teuchos::RCP` used in the C++ version of NOX is hidden from python users. Python versions of C++ methods that accept `Teuchos::RCP<object>` arguments accept raw `object` arguments in python.

The following `NOX` namespaces are suported in python:

```
NOX
NOX.Abstract
NOX.Epetra
NOX.Epetra.Interface
NOX.Solver
NOX.StatusTest
```

The only `NOX` Interface class that has a python wrapper is the `Epetra` interface. As with the C++ version of `NOX`, you define a nonlinear problem by declaring a python class that inherits from `NOX.Epetra.Interface.Required`. Optionally, your class may also inherit from `NOX.Epetra.Interface.Jacobian` and/or `NOX.Epetra.Interface.Preconditioner`.

Your constructor *must* call the constructors of its `NOX.Epetra.Interface` base classes:

```
class MyProblem(NOX.Epetra.Interface.Required):
  def __init__(self):
    NOX.Epetra.Interface.Required.__init__(self)
```

This is necessary because the underlying callback mechanism needs to be properly initialized. The nonlinear function for your class is implemented by defining a `computeF()` method:

```
def computeF(self, x, F, flag):
  "Required implementation of computeF() method"
  ...
```

Arguments `x` and `F` are passed from the underlying C++ code as `Epetra.Epetra_Vector` objects and then converted to `numpy`-hybrid `Epetra.Vector` objects when they are passed into your python `computeF()` function. Therefore, you can use the `numpy.ndarray` interface on these arguments, such as the `shape` attribute, or slice indexing.

The `flag` argument is an integer that `NOX` uses to inform `computeF()` why it is being called. You can use this flag to alter how `computeF()` behaves. See the C++ `NOX` documentation for details.

We are defining a somewhat complicated python-to-C++-to-python callback scheme here. Unfortunately, error-handling in such an environment is not as rubust as in a single-language environment. Specifically, if your `computeF()` (accidentally) raises a python exception, all you may see is:

```
terminate called after throwing an instance of 'Swig::DirectorMethodException'
Abort trap
```

For this reason, it is a good idea to wrap your computations in a `try` block:

```
class MyProblem(NOX.Epetra.Interface.Required):

  def __init__(self):
    NOX.Epetra.Interface.Required.__init__(self)

  def computeF(self, x, F, flag):
    "Required implementation of computeF() method"
    try:
      F[:] = nonlinear_func(x)  # Whatever this may be...
    except Exception, e:
      print "Python exception raised in MyProblem.computeF:"
      print e
      return False
    return True
```

This will print the python exception if one is raised, and tells `NOX` that the computation failed. Remember to return a boolean indicating success or failure of the `computeF()` method.

`NOX` needs to be able to compute the result of the Jacobian of `F(x)` on a given vector from the same space as `x`. The `NOX.Epetra.Interface.Required` class and `computeF()` method are sufficient to estimate this product if you are willing to use the `NOX.Epetra.FiniteDifference` class. If you want to use an algebraic preconditioner based on an approximation to the Jacobian, you will want to use map coloring to greatly improve the efficiency, specifically the `NOX.Epetra.FiniteDifferenceColoring` class. See `exNOX_1Dfem.py` in the `example` directory of the `PyTrilinos` package for an example of this type of implementation.

You solve your problem by first creating an instance of your `MyProblem` class and an `Epetra.Vector` solution (using some appropriate constructor):

```
problem = MyProblem()
soln    = Epetra.Vector(...)
```

Since we do not inherit from `Jacobian` or `Preconditioner`, we need to define a matrix-free linear system for `NOX` to use:

```
mf      = NOX.Epetra.MatrixFree({ }, problem, soln)
fdc     = NOX.Epetra.FiniteDifferenceColoring({ }, { }, soln,
                                              problem.getGraph())
linSys  = NOX.Epetra.LinearSystemAztecOO({ }, { }, mf, mf, fdc, fdc,
                                         soln)
```

Note that these calls assume that your `MyProblem` class defines a `getGraph()` method that returns an `Epetra.CrsGraph` that defines the sparsity/coupling of the nonlinear problem. Also, the empty python dictionaries result in default parameter specifications. Now we can create a `NOX.Epetra.Group` and a `NOX.Solver.Manager`:

```
group   = NOX.Epetra.Group({ }, problem, soln, linSys)
solver  = NOX.Solver.Manager(group, statusTest, { })
```

Here we assume `statusTest` is a properly constructed `NOX.StatusTest` object. Now that everything is specified, we can solve the problem with:

```
status  = solver.solve()
```

Simple, no?

Actually, an attempt has been made to simplify this process in python, with the definition of the following functions:

- `defaultSolver(initGuess, reqInterface, jacInterface=None, jacobian=None, precInterface=None, preconditioner=None, nlParams=None) -> Solver`
- `defaultStatusTest(absTol=None, relTol=None, relGroup=None, updateTol=None, wAbsTol=None, wRelTol=None, maxIters=None, finiteValue=False) -> StatusTest`
- `defaultNonlinearParameters(comm=None, verbosity=0, outputPrec=3, maxIterations=800, tolerance=1.0e-4) -> dict`
- `defaultGroup(nonlinearParameters, initGuess, reqInterface, jacInterface=None, jacobian=None, precInterface=None, preconditioner=None) -> Group`

Depending on the needs of the problem, sometimes you just build the interfaces and call `NOX.defaultSolver()`, which calls the other `default` functions internally. Other times, building the interfaces requires calling the other `default` functions explicitly first. Either way, these functions can significantly reduce the amount of work requird to build a `NOX` solver.

16 PyTrilinos.LOCA

The `LOCA` module is currently disabled. Fortunately, `NOX` has been re-enabled, meaning most of the technical issues related to wrapping `LOCA` have also been addressed.

17 PyTrilinos.Anasazi

The Anasazi Trilinos package makes heavy use of templates, which makes it nearly impossible for the python interface to track it identically. Anasazi classes are templated on one or more of the following parameters:

```
<ScalarType, MV, OP>
```

where `ScalarType` is a primitive data type specifying the floating point representation of the eigenproblem, `MV` is a multi-vector class and `OP` is an operator class. All Anasazi classes are templated on `<ScalarType>`. Some are templated on `<ScalarType, MV>`, and some are templated on all three parameters.

Currently, `PyTrilinos.Anasazi` supports only one interface, to `Epetra`. Thus, the corresponding concrete C++ instantiations are on:

```
<double, Epetra_MultiVector, Epetra_Operator>
```

These are indicated in python by using the Anasazi class name and adding the suffix `Double` for `ScalarType`-only templated classes, or the suffix `Epetra` for any of the other templated classes. So, for example:

```
>>> from PyTrilinos import Anasazi
>>> eig = Anasazi.ValueDouble()
>>> print eig
0+0j
>>> bSort = Anasazi.BasicSortEpetra()
>>> print bSort
<Anasazi.BasicSortEpetra; proxy of <Swig Object of type 'Anasazi::BasicSort<double,
Epetra_MultiVector,Epetra_Operator > *' at 0x6eef50> >
```

An additional layer of abstraction is provided so that these suffixes need not be remembered or used. Since there is currently only one interface, these suffix-less class names are essentially aliases for their longer-named counterparts:

```
>>> eig = Anasazi.Value()
>>> bSort = Anasazi.BasicSort()
>>> print type(eig), type(bSort)
<class 'Anasazi.ValueDouble'> <class 'Anasazi.BasicSortEpetra'>
```

In the future, if additional interfaces are supported (for example, a NumPy interface), then the requested interface will be inferred from the constructor arguments, if possible. For those cases where this is not possible, we will also provide type-specification capabilities that are similar to NumPy.

A second difference between Trilinos Anasazi and `PyTrilinos.Anasazi` is that any methods that expect a

```
Teuchos::RCP<object>
```

as an argument, where `object` is of any type, will take a python object of corresponding type, without the reference-counted pointer wrapper.

The `Anasazi.Eigensolution` class has been changed such that the attributes `Evals`, `Evecs` and `Espace` have been changed to methods `Evals()`, `Evecs()` and `Espace()`. `Evals()` returns a `numpy.ndarray` of type complex double rather than a wrapper to `std::vector<Anasazi::Value<double> >`. Also, the `Evecs()` and `Espace()` methods return `Epetra.MultiVector` objects. Currently, `index` is still an attribute, and is a python list-like object containing integers. (The other attributes must be implemented as methods in order to facilitate the type conversions.)

You can create an `Anasazi.Value` object if you wish, although it is not very functional. It has a `__str__()` method so that you can print its value.

18 PyTrilinos.Thyra

Only preliminary work has been done on the python interface to `Thyra`. It is not an operational package yet.

Generated on: 2010-04-16 18:49 UTC. Generated by Docutils from reStructuredText source.